UNIT 1 GETTING STARTED

Structure

<table>
<thead>
<tr>
<th>Section</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>1.2 An Overview</td>
<td></td>
</tr>
<tr>
<td>1.3 A C Program</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Escape Sequences</td>
<td>12</td>
</tr>
<tr>
<td>1.5 Getting a ‘feel’ for C</td>
<td>14</td>
</tr>
<tr>
<td>1.6 Summary</td>
<td>18</td>
</tr>
<tr>
<td>1.7 Solutions/Answers</td>
<td>18</td>
</tr>
</tbody>
</table>

1.1 INTRODUCTION

In this unit we will introduce you to the C programming language. We begin this unit with an overview of the C programming language, its history and the reasons for its continued popularity in Sec. 1.2.

In Sec. 1.3, we start our discussion of the C programming language with a simple example program. We use the program to tell you how a C program is organised in general.

In Sec. 1.4, we discuss escape sequences which output tab, new line etc. In the last section, we discuss some example C programs to give a feel for the language.

Objectives

After studying this unit, you should be able to

- explain the advantages and disadvantages of C language;
- explain the general structure of C programs; and
- explain the purpose of escape sequences.

1.2 AN OVERVIEW

C is a general purpose computer programming language. It was originally created by Dennis M. Ritchie (circa 1971) for the specific purpose of rewriting much of the Unix operating system. This now famous operating system was at that time in its infancy; it was first written for a discarded DEC PDP-7 computer by Ken Thompson, an electrical engineer and a colleague of Ritchie’s at Bell Labs. Thompson had written the first Unix in a language he chose to call B. B itself was based on BPCL a language that was developed by Martin Richards, another programmer.

The intention of these programmers was to port Unix on to other, architecturally dissimilar machines. Clearly, using any assembly language was out of the question; what they needed was a language that would permit assembly-like (i.e. low or hardware level) operations, such as bit manipulation. At the same time it should be usable on different computers. None of the languages then available served the purpose; they had to create a new one, and C was born. The rest is history; Unix became spectacularly successful, in large measure because of its portability. C, in which most Unix was written, has occupied a pre-eminent position in the development of system programs.

The success enjoyed by Unix, and the consequent popularity of C for systems programming, forced it on the attention of applications programmers, who came to
Introduction to the C Programming Language

appreciate the rich variety of its operators and its control structures. These enable, and in fact encourage, the practice of modular programming: The individual sub tasks that make up a large program can be written in independent blocks or modules, each of manageable size. In other words, the language possesses features that make possible a "divide and conquer" approach to large programming tasks. When programs are organised as modules they are necessarily well-planned, because each module contains only the instructions for a well-defined activity. Such programs are therefore more easily comprehensible by other readers than unstructured programs.

Another desirable quality of modular programs is that they may be modified without much difficulty. If changes are required, only a few modules may be affected, leaving the remainder of the program intact. And last but not least, such programs are easier to debug, because errors can be localised to modules, and removed. For a concrete example, suppose that you have to write a program to determine all prime numbers up to ten million which are of the form \( k^2 + 1 \), where \( k \) is an integer. (2, 5, 17 and 37 are some examples of such primes.) Then you can write one module to generate the number \( k^2 + 1 \) for the next admissible value of \( k \); another module to test whether the last number generated is a prime. Organised in this way, not only will the program be elegant, it would also be easy to debug, understand or modify.

Yet, C is not rigidly structured. The language allows the programmer to forsake the discipline of structured programming, should he or she want to do so.

Another point of difference between C and other languages is in the wealth of its operators. Operators determine the kinds of operation that are possible on data values. The operators of C impart a degree of flexibility and power over machine resources which is unequalled by few other contemporary languages, except assembly language.

But assembly language has disadvantages; it is not structured; it is not easy to learn; moreover, assembly programs for any substantive application tend to be too long; and they are not portable: An assembly language program written for one computer cannot be executed on another with a different architecture and instruction set. On the other hand, a C program created on one computer can be compiled and run on any other machine that has a C compiler.

Possibly the greatest merit of C lies in its intangible quality which can only be termed elegance. This derives from the versatility of its control structures and power of its operators. In practical terms this often means that C programs can be very short, and it must be admitted, sometimes even cryptic. (Indeed, there is a canard that C stands for cryptic!) While brevity is undeniably a virtue (specially when you consider the vastness of COBOL programs!) there can be too much of it. All too frequently in C programs one comes across craftily constructed statements that take as much time to comprehend, as they must have taken to create. In the interest of other readers therefore (as well as, you may find, in your own) you should resist that temptation to be cryptic that C by its nature seems to encourage.

One common classification of computer language divides them into two broad categories: high-level languages (HLLs) and low-level languages (LLLs). An HLL is a language that is easy for a human beings to learn to program in. A good example is BASIC, with its easily understood statements:

```
10 LET X=45  
20 LET Y=56   
30 LET Z=X+Y  
40 PRINT X,Y,Z
50 END
```

An LLL is a language that is closely related to machine or assembly language; it allows a programmer the ability to exploit all of the associated computer's resources. This
power however does not come without a price: LLLs are generally difficult to learn, and are usually so closely tied to the architecture of the host machine that even a person who is familiar with one LLL will have to invest a great deal of effort in learning other. Consider the following statement from the MACRO Assembly Language of VAX series of computers manufactured by the Digital Equipment Corporation, USA:

\[ \text{POPR} \#M<\text{R6, R8, R3}> \]

It is highly unlikely that one who is not familiar with VAX's MACRO Assembler will be able to fathom any meaning from this instruction. Moreover, LLL programs are not portable; they cannot be moved across machines with different architectures.

C is a considerable improvement on LLLs: it's easy to learn; it's widely available (from micros to mainframes); it provides almost all the features that assembly does, even bit-level manipulation (while being fairly independent of the underlying hardware). It also has the additional merit that its programs can be concise, elegant and easy to debug. Finally, because there is a single authority for its definition, there is a greater degree of standardisation and uniformity in C compilers than for other HLLs.

It has been often said with some justification that C is the FORTRAN of systems software. Just as FORTRAN compilers liberated programmers from creating programs for specific machines, the development of C has freed them to write system software without having to worry about the architecture of the machine that the software is intended for. Where architecture-dependent code i.e. assembly code is necessary, it can usually be invoked from within the C environment. C is a middle level language, not as low-level as assembly, and not as high level as BASIC.

In short, C has all the advantages of assembly language, with none of its disadvantages; and it has all the significant features of modern HLLs. It's an easy language to program in, and makes programming great fun.

As we mentioned in the course introduction, we will discuss ANSI C, also known as C89. We will also mention some extra features provided by C99.

We close this section here. In the next section, we will examine some simple C programs to understand the structure of a C program.

1.3 A C PROGRAM

The best way to learn C or any programming language is to begin writing programs in it; so here's our first C program. (See the practical guide for instructions on how to compile and link programs.)

\[
\text{Listing 1: First C program.}
\]

```c
/* Program 1; file name: unitl-progl.c */
#include <stdio.h>
int main(void)
{
    printf("This is easy!!\n");
    return 0;
}
```

There are a few important points to note about this program, points which you will find common to all C programs.

Programs in C consist of functions, one of which must be \texttt{main()}. When a C program is executed, \texttt{main()} is where the action starts. Then other functions may be "invoked" A function is a sub-program that contains instructions or statements to perform a
specific computation on its variables. When its instructions have been executed, the function returns control to the calling program, to which it may optionally be made to return the results of its computations. Because `main()` is a function, too, from which control returns back to the operating system at program termination, in ANSI C it is customary, although not required, to include a statement in `main()` which explicitly returns control to the operating environment. Also, `main()` returns an integer value to the operating system. So, we have added the key word `int` before `main()` statement.

The `int` key word before `main()` is not needed in ANSI C because a function without an explicit return type is assumed to return an integer value. This is not so in C99. If no return type is given, C99 compilers may compile with a warning (as it happens with `gcc`), but it is not compulsory to do so; the program may also fail to compile.

We'll learn more about functions as we go along, but for now you may recognize them by the presence of parentheses after their names.

Apart from `main()`, another function in the program above is the `printf()`. `printf()` is an example of a “library function”. It's provided by the developers of the C library, ready for you to use. C library provides a variety of functions. For example, C library provides many common mathematical functions like the trigonometric functions, exponential function etc. Apart from this, you may use other libraries that provide, for example, graphics functions. These libraries contain the instructions for the function in a compiled form or in the form of object code. So, you do not have to write code in your program for these functions. To enable the use of such libraries, the compilation of C programs is a two part process. In the first part, the program compiles your C file into an object file. The C compiler ‘remembers’ the functions you have called and the linker combines the translated version of the code you wrote with the object code already found in the library. The second step of the process is called linking.

In addition to its variables, a function, including `main()`, may optionally have arguments, which are listed in the parentheses following the function name. The arguments of functions are somewhat different from the arguments that people have: they are values of the parameters in terms of which the function is defined, and are passed to it by the calling program. The instructions which comprise a function are listed in terms of its parameters and its variables.

In the example above, `main()` has no arguments. `printf()` has one: it’s the bunch of characters (more properly: string) enclosed in double quotes:

"This is easy!!
"

When the `printf()` is executed, its built-in instructions process this argument. The result is that the string is displayed on the output device, which we will usually assume is a terminal. The output of the program will be:

**This is easy!!**

with the cursor stationed at the beginning of the next line on the screen. In C a string is not a piece of thread: it’s any sequence of characters enclosed in double quotes. A string may not include a “hard” carriage-return (\<CR\>), i.e. all its characters must be keyed in on a single line on the terminal.

"This is most certainly **not** a C string. \<CR\>
It contains a carriage return Character."

As we have seen in the case of `printf()`, when a function is invoked, it is passed (the values of) its arguments. It then executes its instructions, using these values. On completion the function returns control to the module from which it was invoked. The nice thing about a function is that it may be called as often as required in program.
perform the same computational steps on the different sets of values that are passed to it. Thus there can be several invocations to `printf()` within a program, with different string arguments each time. Also, a function can call other functions. For example, in Listing 1 on page 9, the function `main()` calls the function `printf()` with the text 'This is easy!!' as the argument. In a program for listing primes of the form \(k^2 + 1\) we can have a function that computes \(k^2 + 1\) for the next value of \(k\) passed to it; another function can test whether the value of \(k^2 + 1\) calculated by the first function is a prime or not. These functions could be invoked repeatedly, for different values of \(k\). Functions will be formally introduced in the next Block.

With some compilers you may not require the preprocessor directive:
```
#include <stdio.h>
```
which we have written just before `main()` in Listing 1 on page 9. These directives, unlike other C statements, are not terminated by a semicolon.

Preprocessing is a phase of compilation that is performed prior to the analysis of the program text. We will have more to say about preprocessor directives later.

`stdio.h` is a header file that comes with your C compiler and runtime system and contains data that `printf()` needs in order to output its argument. The `#include` directive causes the inclusion of external text—in this case the file `stdio.h`—in your sources program before the actual compilation proceeds. We'll have more to say about header files later. If you have problems, consult a resident expert!

Let's go back to Listing 1 on page 9. Observe that the body of the program is enclosed in braces {}; a pair of braces defines a block. A program may consist of several blocks, which may include yet other blocks, and so on. The left and right braces which mark a block may be placed anywhere on a line. You will find it easier to read and debug the programs if you align the same column. For the same reason the braces of blocks which are nested inside other blocks are indented a few spaces to the right from the braces of the enclosing block.

```
/* Program 2; file name:unitl-prog2.c */
#include <stdio.h>
int main()
{
    printf("This is the outermost block.\n");
    {
        printf("This is a nested block.\n");
        {
            printf("This block is nested still more deeply.\n");
            printf("This is the innermost block.\n");
        }
    }
    return (0);
}
```

Listing 2: Block structure.

In Listing 2, note that each left brace is balanced by a corresponding right brace. Blocks contain statements such as:
```
printf("This is the innermost block.\n");
```
C statements invariably end with a semicolon. C statements may be of any length and may begin in any column. Each statement may extend over several lines, as long as any included strings are not broken. In C a semicolon by itself constitutes a valid statement, the null statement. Null statements are often very handy, and we will quite frequently have occasion to use them.
While writing a C program, you can add comments, which are ignored by the compiler. Comments in programs are included between a backslash-asterisk pair, /* */:

/* This is a comment about comments. Comments may extend over many lines, but as a rule they should be short. */

Comments are important because they help document a program, and should be written so that its logic becomes transparent. They may be placed wherever the syntax allows "white space" characters: blanks, tabs or newlines.

Do you think comments can be nested, i.e. can you have a comment within a comment? Write a program with a nested comment and see if you can compile it.

In discussion above, we mentioned that a C string cannot contain a carriage return. What should we do if want to break a long string? We have to use an escape sequence. Escape sequences are the topic of our discussion in the next section.

1.4 ESCAPE SEQUENCES

You might be wondering about the \n (pronounced backslash n) in the string argument of the function printf():

"This is easy!!\n"

The \n is an example of an escape sequence: it’s used to print the newline character. If you have executed Programs 1 or 2 you will have noticed that the \n doesn’t appear in the output. Each \n in the string argument of a printf() cause the cursor to be placed at the beginning of the next line of output. Think of an escape sequence as a "substitute character" for printing hard-to-get characters. In an earlier section we learnt that < CR > s are not allowed in strings; but including \n’s within one makes it easy to output it in several lines: if you wish to print a string in two or more lines, place the \n wherever you want to insert a new line in the output, as in the example below:

/* Program 3; file name: unit1-prog3.c */
void main()
{
    printf("This\string\nwill\nbe\nprinted\nin\n9\nlines. \n");
}

Listing 3: Line breaks in a C program.

Each \n will force the cursor to be positioned at the beginning of the next line on the screen, from where further printing will begin. Here is part of the output from executing the above program:

This string will
etc ...

If a string does not contain a \n at its end, the next string to be printed will begin in the same line as the last, at the current position of the cursor, i.e. alongside the first string.

Though < CR > s cannot be included in a string, if you must deal with a long string of characters that cannot fit conveniently in a single line, there is a way of continuing it into the next line. You can insert a backslash(\) followed by a < CR > where you wish to break the string:
“This is a rather long string of characters. It extends over two lines.”

We have seen that the “double quotes” character is used to mark the beginning and end of a C string. How can the “double quotes” character itself be included within a string? This time the escape sequence \" comes to our aid: it prints as ” in the output. Similarly, the escape sequence \\ helps print \, and \’ the single quote character.

All escape sequences in C begin with a backslash.

Here are some exercises to help you check your understanding of escape sequences.

E2) Give the output of the following program:

```c
#include<stdio.h>
int main()
{
    printf("This is the first line of output.");
    printf("But is this the second\nline of output?");
    return (0);
}
```

E3) Execute the program below and obtain the answer to the question in its printf(

```c
#include<stdio.h>
int main()
{
    printf("In how many lines will the output\nof this program be printed?");
    return (0);
}
```

E4) Give the output of the following programs:

a) /* Program 6; file name: unit1-prog6.c */

```c
#include<stdio.h>
int main()
{
    printf("A\n", the teacher said,"is used to\n");
    printf("insert a new line in a C string,\" ");
    printf("IC, IC", said the blind student.\n");
    return (0);
}
```

b) /* Program 7; file name: unit1-prog7.c */

```c
#include<stdio.h>
int main()
{
    printf("To be, or not to be,—");
    printf("that is the question:—\nWhether \"tis ");
    printf("nobler in the mind to suffer\nThe slings ");
    printf("and arrows of outrageous fortune\nOr to ");
    printf("take arms against a sea of troubles,\nAnd--To die,--to sleep,--\n");
    return (0);
}
```

E5) Write a C language program that gives for its output:

```c
/* This is a C comment */
```
Introduction to the C Programming Language

E6) Debug the following program:

```c
#include <stdio.h>

int main()
{
    (print("print this\m" \ *very easy * \).
}
```

Be careful to remember that "" is not the escape sequence for the “double quote” character (as it is in some versions of BASIC). In C "" is the null string; the null string is not “empty”, as one might have thought; it contains a single character, the null character, ASCII 0 (in which all bits are set to zero). We will see later that the last character of every C string is the (invisible) null character. Its presence helps the computer determine the actual end of the string. Other escape sequences available in C are:

- \a Ring terminal bell (the a is for alert) [ANSI extension]
- \? Question mark [ANSI extension]
- \b Backspace
- \r Carriage return
- \f Formfeed
- \t Horizontal tab
- \v Vertical tab
- \0 ASCII null character

Placing any of these within a string causes either the indicated action, or the related character to be output. In the next section, we will see some more C programs that will give you a better idea about the structure of C program.

1.5 GETTING A ‘FEEL’ FOR C

In this section we present a few programs that involve concepts which have not so far been discussed; they’ll be covered at a leisurely pace by and by. As you read these programs, you may discover that many of their statements are self-evident; those that may not be are commented. Create these programs on your computer, compile and execute them, and by the time you come to read about the features used in the programs, you shall have a fair idea about them already.

The major change between ANSI C and the language described in the first edition of K & R is in the declaration and definition of functions. Program 1.11 below is an ANSI C compliant program, but it may not run on your machine if you have a non-ANSI compiler. Program 1.12 is the same program modified to run on Classic C.

So far, all the programs that we have written do the same thing every time we run the program; they print some text, the same text every time. But, we would like to write programs that do different things at different times. For example, suppose we want a program that adds two numbers and gives us the answer. How do we do it? For such programs we need programs that use variables. As the name suggests, variables store values that can change during the run of the program. The next program uses two values, 5 and 7 and stores them in two variables called x and y. If you want the program to do all the various operations for different integers, say 11 and 12, you can edit the source file, change the values of x and y, compile and run the program again. Go ahead! Try out two three different values for x and y. Notice the int key word before the variables x and y. This indicates that x and y will be used to store integer values.

```c
/* Program 8; file name:unitl-prog8.c */
#include <stdio.h>

int main()
{
```
/* Elementary operations with small integers */
/* C calls small integers ints */
int x = 5, y = 7, z;
/* x, y, and z are int variables x is 5, y is 7,
  z doesn't have a value yet; */
printf("x=%d, y=%d\n", x, y);
/* each %d prints a decimal integer. */
z = x - y;
/* now z is x minus y; */
printf("z=x-y=%d\n", z);
/* the * means "multiplied by"; */
printf("z=x*y=%d\n", z);
z = x / y;
/* one int divided by another; */
printf("z=x/y=%d\n", z);
z = y / x;
printf("z=y/x=%d\n", z);
/* guess from the output what
  the % in x % y stands for; */
printf("z=x%y=%d\n", z);
/* To print percent sign
  we use the escape sequence % */
z = y % x;
printf("z=y%x=%d\n", z);
return (0);
}

Now, that wasn't very interesting, isn't it? Will it not be better if the programs asks us
for values of x and y when it is run? We have given below such a program.

/* Program 9; file name: unitl-prog9.c */
#include<stdio.h>
int main()
{
  /* Read values from the keyboard,
     see how scanf () works */
  int x, y, z;
  printf("Enter a value for x.\n");
  printf("Type a small integer, press<CR>: ");
  scanf("%d", &x);
  /* mind that ampersand ", just before x; */
  printf("Enter a value for y: ");
  scanf("%d", &y);
  z = x * y;
  printf("z=x*y=%d\n", z);
  return (0);
}

/* Program 10; file name: unitl-progl0.c */
#include<stdio.h>
int main()
{
  int x, y;
  printf("Enter a value for x: ");
  scanf("%d", &x);
  printf("Enter a value for y: ");
  scanf("%d", &y);
  /* Is x greater than y? Then say so: */
  if (x > y)
    printf("x is greater than y.\n");
  /* else, if it's not, deny it. */
Here is a C program that solves a quadratic equation using the well known formula.
Notice the line

```
#include <math.h>
```

in the program. It loads the definitions of maths library functions. We need this line because we are using the sqrt() function which gives the square root and fabs() that gives the absolute value of a number. This program uses the data type called float to store the absolute value of the coefficients and the discriminant because they may be numbers with a decimal point in them. We will discuss float data type in the next Unit.

```
/* Program to solve a quadratic equation; */
/* File name: unit1-quad.c */
#include <math.h>
#include <stdio.h>
int main()
{
    float a, b, c, disc; /* Use decimals. */
    printf("This program solves the \n the \n quadratic equation ax^2+bx+c=0\n"); /* Continued */
    /* Prompt for the coefficient of the second degree term. */
    printf("Enter the value of a\n");
    /* Read the coefficient of the second degree term */
    scanf("%f", &a);
    /* Stop with an error message if it is 0. */
    if (a == 0) {
```
printf("Value of a should not be zero. Exiting\n");
return 0;
}  
/*If a is not zero, prompt for other
coefficients and read them in.*/
printf("Enter the value of b\n");
scanf("%f", &b);
printf("Enter the value of c\n");
scanf("%f", &c);
printf("You entered a=%f, b=%f, c=%f, a, b, c");
/*Find the discriminant*/
disc = b * b - 4 * a * c;
printf("The discriminant is %f\n", disc);
if (disc == 0) {
    printf("This equation has repeated roots.\n");
    printf("The root is %f with multiplicity 2.\n", -b / (2 * a));
}
if (disc > 0) {
    printf("The roots are real.\n");
    printf("The roots are\n");
    printf("%f", (-b + sqrt(disc)) / (2.0 * a));
    printf("\nand\n");
    printf("%f\n", (-b - sqrt(disc)) / (2.0 * a));
}
if (disc < 0) {
    printf("The roots are complex. The roots are\n");
    printf("%f+I*%f", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
    printf("\nand\n");
    printf("%f-I*%f\n", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
}
return 0;

Here is a C program that performs numerical integration using Simpson's rule. The
program computes \( \int_a^b \frac{dx}{1+x^2} \) by Simpson's rule using 4 sub-intervals. Recall Simpson's
rule: Let \( h = \frac{b-a}{n} \), where \( a \) is the lower limit of integration and \( b \) is the upper limit of
integration and \( n \) is the number of subintervals, which should be even. If we write
\( y_i = f(a+ih) \), then
\[
\int_a^b f(x) dx \approx \frac{h}{3} [y_0 + 4(y_1 + y_3 + \cdots + y_{n-1}) + 2(y_2 + y_4 + \cdots y_{n-2}) + y_n]
\]
Note the definition of the function \( \frac{1}{1+x^2} \) here. Since, \texttt{return} can contain expressions, we
have done the job of finding the value of \( \frac{1}{1+x^2} \) within the return statement. \texttt{return}
statement will simply return the value of the expression. We will see more about
expressions in Unit 3.

/* Simpson's rule; file name:unit1-simpsonf.c*/
#include <stdio.h>
/* function to calculate 1/(1+x*x) */
float f(float x)
{
    return (1 / (1 + x * x));
}

int main()
{
    float integral = 0, h = 1 / 4.0;
    /* Apply Simpson's rule */
    integral =
            (h / 3.0) * (f(0) + 4 * (f(h) + f(3 * h))
+ 2 * f(2 * h))
1.6 SUMMARY

In this unit, we have studied the following points:

1) Because it was written for the purpose of porting Unix system and it was required that C permits low level operations. It also enjoys the advantages of high level languages, being comprehensible. It allows modular programming.

2) The C program is organised into functions and every C program must have a function called main.

3) To print certain special characters we need to use escape sequences:
   \n   New Line
   \' Single quote '
   \" Double quote "
   \ Backslash \n   Ring terminal bell (the a is for alert) [ANSI] extension
   \? Question mark [ANSI extension]
   \b Backspace
   \r Carriage return
   \f Formfeed
   \t Horizontal tab
   \v Vertical tab
   \0 ASCII null character

4) We also examined some example C programs.

1.7 SOLUTIONS/ANSWERS

E1) No, you cannot have nested comments. This is because the compiler will match the first opening /* with the first closing */ and will interpret the rest of the comment as a part of the program. This will result in an error message from the compiler.

E2) This is the first line of output. But is this the second line of output?

E3) In how many lines will the output of this program be printed?

E4) a) "A\n", the teacher said, "is used to insert a new line in a C string."
    "IC, IC", said the blind student.

    b) To be, or not to be,—that is the question—
    Whether 'tis nobler in the mind to suffer
    The slings and arrows of outrageous fortune
    Or to take arms against a sea of troubles,
    And by opposing end them?—To die,—to sleep,—

E5) /*File name:unit1-progl5.c*/
    #include <stdio.h>
    int main()
```c
{    printf("/* This is a C comment */");
    return (0);
}

E6) Following are the mistakes in the program:
   1) Wrong brackets are used around and the stdio.h is also spelt wrongly. It
      should be #include<stdio.h>.
   2) The function name main should not be in capitals. The brackets after main
      should be round brackets and not flower brackets. The correct way is to type
      main ()
   3) The brackets before and after print statement should be flower brackets and
      not round brackets. It should be printf not print. The escape sequence
      \
      is not a legal escape sequence.
   4) The slash in the comment \* This is easy*\ is wrong. It should be a
      backward slash and it should be /*This is easy*/.
```
UNIT 2 DATA TYPES IN C

2.1 INTRODUCTION

Computer programs work with data. For this data has to be stored in the computer and manipulated through programs. To facilitate this, data is divided into certain specific types. Most modern programming languages are to an extent typed; i.e., they admit only of particular, pre-declared types of data or variables in their programs, with restrictions on the kinds of operations permissible on a particular type. Typing helps eliminate errors that might arise in inadvertently operating with unrelated variables. Another benefit of typing is that it helps the compiler allot the appropriate amount of space for each program variable: one byte for a character, two for an integer, four for a real, etc. C too is a typed language, but we shall find that it is not as strongly typed as, for example, Pascal is. Thus, it provides the programmer with a far greater degree of flexibility. At the same time you should not forget that with this greater power inherent in language, the C programmer shoulders all the more responsibility for writing code that shall be robust and reliable in all circumstances.

We start this unit with discussion of the data type int in Sec. 2.2. As you can easily guess, this is used for storing data which are integers like number of students in a class etc. In Sec. 2.3, we discuss the data type char which are used to hold characters. In Sec. 2.4, we will discuss float which are used to hold decimal numbers like decimal approximations to π, √2 etc. In Sec. 2.5, we will discuss double. They are also used to store decimal numbers, but they can store larger numbers with greater accuracy. The enum and typedef, which we discuss in sections 2.6 and 2.7, respectively, helps us to create user defined data types.

Objectives

After studying this unit, you should be able to

- state what the basic data types are, how they are declared and how they are used;
- explain the use of modifiers unsigned, double and long;
- create simple MACRO definitions and correctly name the identifiers according to the rules;
- use scanf() and printf() to read and print variables of different data types; and
- use the typedef statement and explain its purpose.
2.2 VARIABLES OF TYPE int

We have already seen that C uses different data types for storing integers and numbers with a decimal digit. Actually, C program variables and constants are of four main types: char, int, float and double. We start our discussion of data types with the discussion of the data type int.

Before an identifier can be used in a C program its type must be explicitly declared. Here's a declarative statement of C:

```c
int apples;
```

This statement declares the programmer’s intent that apples will store a signed integer value, i.e., apples may be either a positive or a negative integer within the range set for variables of type int. Now this range can be very much machine dependent; it depends, among other things on the word size of your machine. For most old DOS compilers for the IBM PC for which the word size is 16 bits ints are stored in two consecutive bytes, and are restricted to the range \([-32768,32767]\). In most of the modern compilers which are 32 bit ints are 4-bytes signed integers in the range \([-2147483648,2147483647]\). In declaring apples to be an int you are telling the compiler how much space to allocate for its storage. You have also made an assumption of the range in which you expect its value to lie.

In C it is possible and in fact usual to both declare a variable’s type and, where needed, initialise its value in the same statement:

```c
int salary = 5000;
```

It is not correct to assume that a variable which has only been declared e.g.:

```c
int volts; /*volts is unpredictable*/
```

but has not been initialised, i.e. assigned a value, automatically gets the value 0.

Let's look at Listing 1, and its output. The program adds two ints x and y, and prints their sum, z. Recall that the printf() can be used to print numbers just as easily as it prints strings. To print an int x the following printf() will do:

```c
printf("The value of x is: %d\n", x);
```

The %d signifies that x is to be printed as a decimal integer.

Listing 1: A simple program that uses int.

```c
/* Program 1; file name: unit2-prog1.c */
#include <stdio.h>
int main()
{
    int x = 5, y = 7, z;
    z = x + y;
    printf("The value of x is: %d\n", x);
    printf("The value of y is: %d\n", y);
    printf("Their sum, z, is: %d\n", z);
    return (0);
}
```

The output of the program in Listing 1 is appended below.

The value of x is: 5
The value of y is: 7
Their sum, z, is: 12

To understand the very great importance of using variables in a computation only after having assigned them values, execute the program in Listing 2 on the facing page and determine its output on your computer:
Program 2; file name: unit2-prog2.c

```c
#include <stdio.h>

int main()
{
    int x, y, z;  /* x, y and z are undefined. */
    z = x + y;
    printf("The value of x is: %d\n", x);
    printf("The value of y is: %d\n", y);
    printf("Their sum, z is: %d\n", z);
    return (0);
}
```

Listing 2: When variables are not defined.

On our 32-bit linux machine, the output was:

**The value of x is:** 134513628

**The value of y is:** -1208279840

**Their sum, z is:** -1073766212

You may like to compile the program, run it and check the output on your computer.

Now, looking at the output of this program, could you possibly have predicted the values x, y and z would get? **Moral:** Never assume a variable has a meaningful value, unless you give it one.

Try the following exercises now.

---

**E1)** Execute the program below, with the indicated arithmetic operations, to determine their output:

```c
/* Program 3; file name: unit2-prog3.c */
#include <stdio.h>

int main()
{
    int x = 70, y = 15, z;
    printf("x=%d,\n", x);
    printf("y=%d,\n", y);
    z = x - y;
    printf("Their difference x-y is: %d\n", z);
    z = x * y;
    printf("Their product x*y is: %d\n", z);
    z = x / y;
    printf("The quotient x/y is: %d\n", y);
    return (0);
}
```

Can you explain your results? What do you think is happening here? In particular, determine the values you obtain for the quotient x/y when:

- x = 60, y = 15;
- x = 70, y = 15;
- x = 75, y = 15;
- x = 80, y = 15;

**E2)** Execute Program 4 below for the stated values of x and y. What mathematical operation could the % symbol in the statement:

```
z=x%y;
```

signify?
```c
/* Program 4; file name: unit2-prog4.c */
#include <stdio.h>
int main()
{
    int x = 60, y = 15, z;
    printf("x=%d, \n", x);
    printf("y=%d, \n", y);
    z = x % y; /* What does the % operator do? */
    printf("z is: %d\n", z);
    x = 70;
    y = 15;
    z = x % y;
    printf("z is: %d\n", z);
    x = 75;
    y = 15;
    z = x % y;
    printf("z is: %d\n", z);
    x = 15;
    y = 80;
    z = x % y;
    printf("z is: %d\n", z);
    return (0);
}
```

### 2.2.1 Range Modifiers for int Variables

On occasions you may need to work with strictly non-negative integers, or with integers in a shorter or longer interval than the default for ints. The following types of range modifying declarations are possible:

**unsigned**

```c
unsigned int variable-name;
```

**Usage**

**Example:** `unsigned int stadium_seats;`

This declaration "liberates" the sign bit, and makes the entire word (including the freed sign bit) available for the storage of non-negative integers.

**Note**

The sign bit—the leftmost bit of a memory word—determines the sign of the contents of the word; when it's set to 1, the value stored in the remaining bits is negative. Most architectures use two's complement arithmetic, in which the sign bit is "weighted", i.e. it has an associated place value which is negative. Thus on a 16-bit machine its value is $-2^{15}$, or $-32,768$. So a 16-bit signed number such as 10000000 00111111 would have the value $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 - 2^{15} = -32,705$. As an unsigned integer this string of bits would have the value 32831.

On PCs the unsigned declaration allows for int variables the range at least [0, 65535] and is useful when one deals with quantities which are known beforehand to be both large and non-negative, e.g. memory addresses, a stadium's seating capacity, etc.

Just as `%d` in the `printf()` prints decimal int values `%u` is used to output unsigned ints, as the program in Listing 3 on the next page illustrates. Execute the program and determine its output; also examine the effect of changing the `%u` format conversion specifiers to `%d` in the `printf()`s. Can you explain your results?
typedef short int variable-name; 
Example: short int friends;

The short int declaration may be useful in instances where an integer variable is known beforehand to be small. The declaration above ensures that the range of friends will not exceed that of ints, but on some computers the range may be shorter (e.g. -128 through 127); friends may be accommodated in a byte, thus saving memory. There was a time in the olden days of computing, when main memory was an expensive resource, that programmers tried by such declarations and other stratagems to optimise core usage to the extent possible. But for present-day PCs, with memory cheap and plentiful, most compiler writers make no distinction between ints and short ints.)

The %d specification in the printf() is also used to output short ints.

unsigned short

typedef unsigned short int variable-name; 
Example: unsigned short int books;

The range of books will not exceed that of unsigned ints; it may be shorter.

long

typedef long int variable-name; 
Example: long int stars_in_galaxy;

This declaration is required when you need to use integers larger than the default for ints. On most computers long ints are 4-byte integers ranging over the interval [-2147483648,2147483647]. When a long integer constant is assigned a value the letter L (or l) must be written immediately after the rightmost digit:

long int big_num = 1234567890L;

%ld in the printf() outputs long decimal ints, as you may verify by executing the program in Listing 4:

/* Program 6; file name:unit1-prog6.c */
#include<stdio.h>
int main()
{ long int population_2020 = 424967295L;
 printf("The population of this country in 2020 AD\n");
 printf("will exceed %ld if we do\n", population_2020);
 printf("not take remedial steps now.\n");
 return (0);
}

Listing 4: Format modifier for printing long integers.

unsigned long

Usage

unsigned long int variable-name;
Example: usage: unsigned long int population_2020;

The unsigned long declaration transforms the range of long ints to the set of 4-byte non-negative integers. Here population_2020 is allowed to range over [0, 4294967295], (Let's hope that larger-sized words will not be required to store this value!) unsigned longs are output by the %lu format conversion specifier in the printf().

In the above declarations shorter forms are allowed: thus you may write:

unsigned letters;  /* instead of unsigned int letters; */
long rope;        /* instead of long int rope; */
unsigned short note;  /* instead of unsigned short int note; */
etc.

E3) State the output from the execution of the following C program:

/* Program 7; file name: unitl-prog7.c */
#include<stdio.h>
int main()
{
    unsigned cheque = 54321;
    long time = 1234567890L;  /* seconds */
    printf("I've waited a long\n
time (%ld seconds)\n", time); /*Continued*/
    printf("for my cheque (for Rs.%u/-),\nand now\n", cheque); /*Continued*/
    printf("I find it\'s unsigned!\n");
    return (0);
}

Modify this program appropriately to convert the time in seconds to a value of days, hours, minutes and seconds so that it gives the following additional output on execution:

That was a wait of:

14288 days,
23 hours,
31 minutes
and 30 seconds.

(Hint: If a and b are int variables, then a/b is the quotient and a % b the remainder after division of a by b if b is less than a.)

As remarked above, on most current compilers for the PC the range of shorts is not different from the ints; similarly the range of unsigned shorts is the same as that of unsigned ints.

Thus we have:
More than one variable can be declared in a single statement:

```c
int apples, pears, oranges, etc;
unsigned long radishes, carrots, lotus_stems;
```

Consider the statement:

```c
int x = 1, y = 2, z;
```

This declares `x` to be an int with value 1, `y` to be an int with value 2, and `z` to be an int of unpredictable value. Similarly, the statement:

```c
int x, y, z = 1;
```

declares `x`, `y` and `z` to be ints. `x` and `y` are undefined. `z` has the value 1.

Octal (base 8) or hexadecimal (base 16) integers may also be assigned to int variables; octal and hexadecimal integers are used by assembly language programmers as shorthand for the sequences of binary digits that represent memory addresses or the actual contents of memory locations. (That C provides the facility for computing with octal and hexadecimal integers is a reminder of its original raison d'être: systems programming). An octal integer is prefaced by a 0 (zero), a hexadecimal by the letters Ox or OX, for example

```c
int p = 012345, q = 0x1234; /*p is in octal notation, q is in hex */
long octal_num = 012345670L, hex_num = 0X7BCDEF89L;
```

The printf() can be used to output octal and hexadecimal integers just as easily as it prints decimal integers: %0 prints octal ints, while %x (or %X prints hexadecimal ints; long octal or hex ints are printed using %lo and %lx, respectively, %#0 and %#x (or %#X) cause octal and hexadecimal values in the output to be preceded by 0 and by 0x (or 0X) respectively. (A lowercase x outputs the alphabetical hex digits in lowercase, an uppercase X outputs them in uppercase characters.) Precisely how format control may be specified in a printf() is described in the next Unit.

What are the decimal values of `p`, `q`, `octal_num` and `hex_num`?

One easy way of verifying the answer to the last question is to get printf() to print the numbers in decimal, octal and hexadecimal notation:

```c
printf ("As an octal number, hex_num = %lo\n", x);
printf ("As a decimal number, octal_num = %ld\n", x);
printf ("As a hexadecimal number, octal\num = %lx\n", x);
```

What happens if in the course of a computation an int or int like variable gets a value that lies outside the range for the type? C compilers give no warning. The computation proceeds unhampered. Its result is most certainly wrong. This is in contrast to Pascal where overflow causes the program to be aborted. The program in Listing 5 on the next page of exercise 5 demonstrates this. There is also an exercise to test your understanding of format modifiers for printing hexadecimal and octal numbers. Please try them.

---

E4) Print the values of `p`, `q`, `octal_num` and `hex_num` using each of the %#0, %#l0, %#lx and %##lX format conversion characters respectively.

E5) Execute the program in Listing 5 on the next page (in which a variable `x` is multiplied by itself several times), and determine its output on your machine:
/* Program 8: file name: unit2-prog8.c */
#include <stdio.h>

int main()
{
    int x = 5;
    printf("x = %d\n", x);
    x = x * x;               /* now x is 5*5 =25 */
    printf("x = %d\n", x);
    x = x * x;               /* now x exceeds 
                            the limit of int on Old 16 bit compilers. */
    printf("x = %d\n", x);
    x = x * x;               /* now x exceeds 
                            the limit of int in 32 bit machines also */
    printf("x = %d\n", x);
    return (0);
}

Listing 5: Example demonstrating the effects of overflow.

One statement that often confuses novice programmers is:

    x = x*x;

If you have studied algebra, your immediate reaction may well be: "This can't be right, unless x is 0 or x is 1; and x is neither 0 nor 1 in the program.!” True; we respect your observation; however, the statement:

    x = x*x;

is not an equation of algebra! It's an instruction to the computer, which in English translates to the following:

Replace x by x times x.

Or, more colloquially, after its execution:

(new value of x) is (old value of x)*(old value of x).

That's why in Program 8, exercise 5 on the preceding page, x begins with the value 5, then is replaced by 5*5 (which is 25) then by 25*25 (which is 625), and then by 625*625, which is too large a value to fit inside 16 bits.

In ANSI C, a decimal integer constant is treated as an unsigned long if its magnitude exceeds that of signed long. An octal or hexadecimal integer that exceeds the limit of int is taken to be unsigned; if it exceeds this limit, it is taken to be long; and if it exceeds this limit it is treated as unsigned long. An integer constant is regarded as unsigned if its value is followed by the letter u or U, e.g. 0x9999u; it is regarded as unsigned long if its value is followed by u or U and 1 or L, e.g. 0xfffffffful.

The system file limits.h available in ANSI C compliant compilers contains the upper and lower limits of integer types. You may include it before main() precisely as your #include <stdio.h>:

    #include <limits.h>

and thereby give to your program access to the constants defined in it. For example, the declaration:

    long largest = LONG_MAX;
Table 1: Limits for int types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>bits in a char</td>
<td>8</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>maximum value of char</td>
<td>UCHAR_MAX or SCHAR_MAX</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>minimum value of char</td>
<td>0 or SCHAR_MIN</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>maximum value of int</td>
<td>32767</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>minimum value of int</td>
<td>-32767</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>maximum value of long</td>
<td>2147483647</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>minimum value of long</td>
<td>-2147483647</td>
</tr>
<tr>
<td>SCHAR_MAX</td>
<td>maximum value of signed char</td>
<td>127</td>
</tr>
<tr>
<td>SCHAR_MIN</td>
<td>minimum value of signed char</td>
<td>-127</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>maximum value of short</td>
<td>32767</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>minimum value of short</td>
<td>-32767</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>maximum value of unsigned char</td>
<td>255</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>maximum value of unsigned int</td>
<td>65535</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>maximum value of unsigned long</td>
<td>4294967295</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>maximum value of unsigned short</td>
<td>65535</td>
</tr>
</tbody>
</table>

will initialise largest to 2147483647. The values stated below are accepted minimum magnitudes. Larger values are permitted. Let us now write a program that uses limits.h. The program in Listing 6 prints the maximum and minimum values of int and the maximum value of unsigned int. (What is the minimum value of unsigned int?)

/*Program to print limits; file name: printlimits.c*/
#include <stdio.h>
#include <limits.h>

int main()
{
    printf("Maximum value of int is \%d: \n", INT_MAX);
    printf("Minimum value of int is \%d: \n", INT_MIN);
    printf("Maximum value of unsigned is \%u: \n", UINT_MAX);
    return (0);
}

Listing 6: Program that prints limits.

Here is an exercise for you to check for yourself whether you can use the values in limits.h file.

E6) Modify the program so that it prints the limits for the other types in Table 1.

Compile and run the program on your computer and see what are the values you get.

You will agree that the programs we’ve been working with would be much more fun if we could supply them values we wished to compute with, while they’re executing, rather than fix the values once and for all within the programs themselves, as we’ve been doing. What we want, in short, is to make the computer scan values for variables from the keyboard. When a value is entered, it’s assigned to a designed variable. This is the value used for the variable in any subsequent computation. Program 9 below illustrates how this is done. It accepts two numbers that you type in, and adds them. It uses the scanf() function, the counterpart of the printf() for the input of data. We have already seen an example in Unit 1, that uses scanf() function. Here we will give some more details about this function. We will discuss both this function and printf() in greater detail later. But, note one important difference: when scanf() is to read a variable x, we write scanf("\%d", &x);
In `scanf()`, in contrast to `printf()`, the variable name is preceded by the
ampersand, &. In the next Unit we will learn that placing the & character before a
variable's name yields the memory address of the variable. Quite reasonably a variable
that holds a memory address is called a *pointer*. It points to where that variable can be
found. Variables which can store pointers are called *pointer variables*. Where
`printf()` uses variable names, the `scanf()` function uses pointers.

Experiment with the program in Listing 7 to see what happens if any of x or y or z
exceed the limit of `ints` for your computer.

```c
/* Program 9; file name: unit2-prog9.c */
#include <stdio.h>
int main()
{
    int x, y, z;
    printf("Type in a value for int x, and press <CR>: ");
    scanf("%d", &x);
    printf("\n is: %d\n\n", x);
    printf("Type in a value for int y, and press <CR>: ");
    scanf("%d", &y);
    printf("\n is: %d\n\n", y);
    z = x + y;
    printf("The sum of x and y is %d\n", z);
    return (0);
}
```

Listing 7: Experimenting with limits of `ints`.

E7) Write a program that reads two non-negative integers of type `unsigned long` and
cchecks if their sum entered exceeds the limit for `unsigned long`.

We end this section here. In the next section we will discuss another type of variable
that can be used to store characters.

### 2.3 VARIABLES OF TYPE `char`

Character variables are used to store single characters for the ASCII set. They're
accommodated in a single byte. Character variables are declared and defined as in the
statement below:

```c
char bee = 'b', see = 'c', ccc;
```

This declaratory statement assigns the *character constant* ‘b’ to the *char* variable
bee, and the character constant ‘c’ to the *char* variable named see. The *char*
variable named see. The *char* variable ccc is undefined.

Character constants are single characters. They must be enclosed in right single quotes.
Escape sequences may be also assigned to character variables in their usual backslash
notation, with the "compound character" enclosed in right single quotes. Thus the
statement:

```c
char nextline = '\n';
```

assigns the escape sequence for the *newline* character to the *char* variable
nextline.

---

**Note**

In ANSI C a character constant is a sequence of one or more characters en-
closed in single quotes. Precisely how the value of such a constant is to be
interpreted is left to the implementation.
Since character variables are accommodated in a byte, C regards chars as being a subrange of ints, (the subrange that fits inside a byte) and each ASCII character is for all purposes equivalent to the decimal integer value of the bit picture which defines it. Thus 'A', of which the ASCII representation is 01000001, has the arithmetical value of 65 decimal. This is the decimal value of the sequence of bits 01000001, as you may easily verify. In other words, the memory representation of the char constant 'A' is indistinguishable from that of the int constant, decimal 65. The upshot of this is that small int values may be stored in char variables, and char values may be stored in int variables! Character variables are therefore signed quantities restricted to the range [−128, 127]. However, it is a requirement of the language that the decimal equivalent of each of the printing characters be non-negative. We are assured then that in any C implementation in which a char is stored in an 8-bit byte, the corresponding int value will always be a non-negative quantity, whatever the value of the leftmost (sign) bit may be. Now, identical bit patterns within a byte may be treated as a negative quantity by one machine, as a positive by another. For ensuring portability of programs which store non-character data in char variables the unsigned char declaration is useful: it changes the range of chars to [0, 255].

Note

However, the ANSI extension signed char explicitly declares a signed character type to override, if need be, a possible default representation of unsigned chars.

One consequence of the fact that C does not distinguish between the internal representation of byte-sized ints and chars is that arithmetic operations which are allowed on ints are also allowed on chars! Thus in C, if you wish to, you may multiply one character value by another. I list the ASCII decimal equivalents of the character set of C. Here are some variables declared as chars, and defined as escape sequences:

```c
char newline = '\n', single_quote = '\"';
```

Character constants can also be defined via their octal ASCII codes. The octal value of the character, which you may find from the table in Appendix I, is preceded by a backslash, and is enclosed in single quotes:

```c
char terminal_bell = \07'; /* 7=octal ASCII code for beep */
char backspace = \010'; /* 10=octal code for backspace */
```

Note

For ANSI C compilers, character constants may be defined by hex digits instead of octals. Hex digits are preceded by x, unlike 0 in the case of octals. Thus is ANSI C:

```c
char backspace = '\xA';
```

is an acceptable alternative declaration to

```c
char backspace = '\010';
```

Any number of digits may be written, but the value stored is undefined if the resulting character value exceeds the limit of char.

On an ASCII machine both 'b' and '010' are equivalent representations. Each will print the backspace character. But the latter form, the ASCII octal equivalent of 'b', will not work on an EBCDIC machine, typically an IBM mainframe, where the collating sequence of the characters (i.e., their gradation or numerical ordering) is different. In the interests of portability therefore it is preferable to write 'b' for the backspace character, rather than its octal code. Then your program will work as certifiably on an EBCDIC machine as it will on an ASCII.
Introduction to the C Programming Language

Note that the character constant 'a' is not the same as the string "a". (We will learn later that a string is really an array of characters, a bunch of characters stored in consecutive memory locations, the last location containing the null character; so the string "a" really contains two char, 'a' immediately followed by '\0'). It is important to realise that the null character is not the same as the decimal digit 0, the ASCII code of which is 00110000.

Just as %d in printf() or scanf() allows us to print and read ints, %c enables the input and output of single characters which are the values of char variables. Let's look at Programs 2.10 and 2.11 below:

Program 2.10; file name: unit2-progl0.c */
#include <stdio.h>
int main()
{
 char a = 'H', b = 'e', c = 'l', d = 'o', newline = '\n';
 printf("%c", a);
 printf("%c", b);
 printf("%c", c);
 printf("%c", d);
 printf("%c", newline);
 return (0);
}
The output of Program 2.10 is easily predictable (what is it?).

Program 2.11; file name: unit2-progl1.c */
#include <stdio.h>
int main()
{
 char val_1 = 'a', val_2 = 'z';
 int val_3;
 printf("The first character is %c \n and its decimal equivalent is %d.\n", val_1, val_1);/*Continued*/
 printf("The second character is %c \n and its decimal equivalent is %d.\n", val_2, val_2);/*Continued*/
 val_3 = val_1 * val_2;
 printf("Their product is %d\n", val_3);
 return (0);
}

Execute the program above to verify that the char variables behave like one byte ints in arithmetic operations.

The %c format conversion character in a printf() outputs escape sequences, as you saw in Program 2.10. Execute Program 2.12 if you want a little music:

Program 2.12; File name: unit2-progl2.c */
#include <stdio.h>
int main()
{
 char bell = '\007'; /* octal code of the terminal bell */
 char x = 'Y', y = 'E', z = 'S', exclam = '!';
 printf("Do you hear the bell? %c%c%c%c%c%c", bell, x, bell, y, bell, z, exclam);/*Continued*/
 return (0);
}

Program 2.13 assigns a character value to an int variable, an integer value to a char variable, and performs a computation involving these variables. Predict the output of the program, and verify your result by executing it.

Program 2.13; file name: unit2-progl3.c*/
#include <stdio.h>
int main()
{
    int alpha = 'A', beta;
    char gamma = 122;
    beta = gamma - alpha;
    printf("beta seen as an int is: %d\n", beta);
    printf("beta seen as a char is: %c\n", beta);
    return (0);
}

One character that is often required to be sensed in C programs is not strictly speaking a character at all: it's the EOF or End-Of-File character, and its occurrence indicates to a program that the end of terminal or file input has been reached. Because the EOF is not a character, being outside the range of chars, any program that's written to sense it must declare an int variable to store character values. As we see in Program 13, this is always possible to do. An int variable can accommodate all the characters assigned to it, and can also accommodate EOF. We'll see uses for EOF in later units.

The putchar() function (pronounced “put character”), which takes a character variable or constant as its sole argument, is often a more convenient alternative for screen output than is the printf(). When this function is invoked (for which purpose you may need to #include<stdio.h>), the character equivalent of its argument is output on the terminal, at the current position of the cursor:

putchar (char_var);

Suppose char_var has been assigned the value 'A'. Then 'A' will be displayed where the cursor was.

Reciprocally, getchar() (pronounced “get character”) gets a single character from the keyboard, and can assign it to a char variable. (stdio.h may have to be #included before getchar() can be used.) It has no arguments, and is typically invoked in the following way:

char_var = getchar();

When such a statement is encountered, the execution of the program is stayed until a key is pressed. Then char_var is assigned the character value of the key that was pressed.

Program 14 below illustrates the usage of these functions. Your compiler may require the inclusion of stdio.h to invoke getchar() and putchar().

/* Program 14; file name:unit2-prog14.c */
#include<stdio.h>
int main()
{
    char key_pressed;
    printf("Type n a lowercase letter (a-z), press <CR>: ");
    key_pressed = getchar();  /* get char from keyboard */
    printf("You pressed ");
    putchar(key_pressed - 32);
    /* put uppercase char on Terminal */
    putchar('\n');
    /* converts to uppercase because
     ASCII decimal equivalent is 32
     less than for the corresponding
     lowercase character. */
    return (0);
}
There are certain characters required in C programs which are not available on the keyboards of many non-ASCII computers. In ANSI C these characters can be simulated by trigraph sequences, which are sequences of three characters of the form ??x. They're treated as special characters even if they are embedded inside character strings.

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>Substitutes for</th>
</tr>
</thead>
<tbody>
<tr>
<td>?? =</td>
<td>#</td>
</tr>
<tr>
<td>??(</td>
<td>[</td>
</tr>
<tr>
<td>??)</td>
<td>]</td>
</tr>
<tr>
<td>?? &lt;</td>
<td>{</td>
</tr>
<tr>
<td>?? &gt;</td>
<td>}</td>
</tr>
<tr>
<td>??/</td>
<td>\</td>
</tr>
<tr>
<td>??'</td>
<td>'</td>
</tr>
<tr>
<td>??!</td>
<td></td>
</tr>
<tr>
<td>??-</td>
<td>-</td>
</tr>
</tbody>
</table>

Here are some exercises to give you some practice in working with char.

E8) In Program 13 insert the declaration:
unsigned char delta = alpha-gamma;
and print the values of delta as an int, an unsigned int, a char and as an unsigned char quantity.

E9) Write a program which gets a character via getchar(), and prints its ASCII decimal equivalent.

We close this section here. In the next section, we will take up another data type called float that is used for storing fractions and irrational numbers.

2.4 VARIABLES OF TYPE float

Integer and character data types are incapable of storing numbers with fractional parts. Depending on the precision required, C provides two variable types for computation with “floating point” numbers, i.e. numbers with a decimal (internally a binary) point. Such numbers are called floats because the binary point can only notionally be represented in the binary-digits expansion of the number, in which it is made to “float” to the appropriate “position” for optimal precision. (You can immediately see the difficulty of imagining a binary “point” within any particular bit of a floating point word, which can contain only a 0 or a 1!) Typically, some of the leftmost bits of a floating point number store its characteristic (the positive or negative power of two to which it is raised), and the remaining its mantissa, the digits which comprise the number. In base 10, for example, if 2.3 is written as 0.0023 \times 10^3, the mantissa is 0.0023, and the characteristic (exponent) is 3.

Single precision floating point variables are declared by the float specification, for example:

```
float bank_balance = 1.234567E8;
/* En means 10 raised to the power n*/
```

The scientific notation En, where the lowercase form en is also acceptable, is optional; one may alternatively write:

```
float bank_balance = 123456700.0;
```
Floats are stored in four bytes and are accurate to about seven significant digits; on PCs their range extends over the interval \([E^{-38}, E^{37}]\).

It must never be lost sight of that the floating point numbers held in a computer's memory are at best approximations to real numbers. There are two reasons for this shortcoming. First, the finite extent of the word size of any computer forces a truncation or round-off of the value to be stored; whether a storage location is two bytes wide, or four, or even eighth, the value stored therein can be precise only to so many binary digits.

Second, it is inherently impossible to represent with unlimited accuracy some fractional values as a finite series of digits preceded by a binary or decimal point. For example

\[ 1/7 = 0.142857142857142857... \text{ ad infinitum} \]

As long as a finite number of digits is written after the decimal point, you will be unable to accurately represent the fraction 1/7. But rational fractions that expand into an infinite series of decimals aren't the only types of floating point numbers that are impossible to store accurately in a computer. Irrational numbers such as the square root of 2 have aperiodic (non-repeating) expansions—there's no way that you can predict the next digit, as you could in the expansion of 1/7 above, at any point in the series. Therefore it's inherently impossible to store such numbers infinitely accurately inside the machine. The number \( \pi \), which is the ratio of the circumference of any circle to its diameter, is another number that you cannot represent as a finite sequence of digits. As you know it's not merely irrational, it's a transcendental number. (It cannot be the root of any algebraic equation with rational coefficients.) So, an element of imprecision may be introduced by the very nature of the numbers to be stored.

Third, in any computation with floating point numbers, errors of round-off or truncation are necessarily introduced. For, suppose you multiply two \( n \)-bit numbers; the result will in general be a \( 2n \)-bit number. If this number \( (2n) \) of bits is larger than the number of bits in the location which will hold the result, you will be forcing a large object into a small hole! Ergo, there'll be a problem! Some of it will just have to be chopped off. Therefore it is wisest to regard with a pinch of salt any number emitted by a computer as the result of computation, that has a long string of digits after the decimal point. It may not be quite as accurate as it seems.

The \%e, \%f and \%g format conversion characters are used in the `scanf()` and `printf()` functions to read and print floating point numbers \%e (or \%E) is used with floating point numbers in exponent format, while \%g(or \%G) may be used with floats in either format, \%g (or \%G) in the `printf()` outputs a float variable either as a string of decimal numbers including a decimal point, or in exponent notation, whichever is shorter. An uppercase E or G prints an uppercase E in the output.

We shall be discussing more about format control in a later Unit.) Program 15 below finds the average of five numbers input from the keyboard, and prints it:

```c
/* Program 15; file name:unit2-prog15.c */
#include <stdio.h>
int main()
{
    float val_1, val_2, val_3, val_4, val_5, total = 0.0, avg;
    printf("\nEnter first number...");
    scanf("%f", &val_1);
    printf("\nEnter second number...");
    scanf("%f", &val_2);
    printf("\nEnter third number...");
    scanf("%f", &val_3);
    printf("\nEnter fourth number...");
    scanf("%f", &val_4);
    printf("\nEnter fifth number...");
    scanf("%f", &val_5);
    total = val_1 + val_2 + val_3 + val_4 + val_5;
    avg = total / 5;
    printf("\nThe average is \%.2f\n", avg);
    return 0;
}
```

Data Types in C
Here's a sample conversation with the program:

Enter first number... 32.4
Enter second number... 56.7
Enter third number... 78.3
Enter fourth number... 67.8
Enter fifth number... -93.9

The average of the numbers you entered is: 28.260000

You can check your understanding of our discussion of floats by doing the following exercise.

E10) Write a program to compute simple interest: If a principal of P Rupees is deposited for a period of T years at a rate of R per cent, the simple interest I is given by:

\[ I = \frac{PRT}{100} \]

Your program should prompt for floats P, R and T from the keyboard and output the interest I.

We end this section here. In the next section, we discuss double, a data type for storing larger floating numbers.

2.5 VARIABLES OF TYPE double

Because the words of memory can store values which are precise only to a fixed number of figures, any calculation involving floating point numbers almost invariably introduces round-off errors. At the same time scientific computations often demand a far greater accuracy than that provided by single precision arithmetic, i.e., arithmetic with the four-byte float variables. Thus, where large scale scientific or engineering computations are involved, the double declaration becomes the natural choice for program variables. The double specification allows the storage of double precision floating point numbers (in eight consecutive bytes) which are held correct to 15 figures, and have a much greater range of definition than floats. [E-308, E307]. Older compilers may also allow the long float specification instead of double, but its use is not recommended.

\[
\text{double \ lightspeed} = 2.997925 \times 10^{10}, \ \text{pi} = 3.1415928;
\]

The %lf (or %le or %lg) specification in a scanf() is required for the input of double variables, which are however output via %e (or %E), %f or %g like their single precision counterparts. Program 16 which computes the volume of a cone, whose base radius and height are read off from the keyboard, illustrates this:

```c
#include <stdio.h>
#define PI 3.1415928

int main()
{
    double base_radius, height, cone_volume;
    printf("This program computes the volume of a cone\n");
    printf("of which the radius and height are entered\n");
```
printf("from the keyboard. \n\nEnter radius of cone base: ");  
scanf("%lf", &base_radius);  
printf("\nEnter height of cone: ");  
scanf("%lf", &height);  
printf("\nVolume of cone of base radius R and height H is (1/3)*PI*R*R*H \n\Continued/ 
cone_volume = PI * base_radius * base_radius * height / 3;
printf("\nVolume of cone is: %f\n", cone_volume);
return (0);
}

One new feature of Program 16 is to be found in its second line.

#define PI 3.1415928

The #defines—also called MACRO definitions—are a convenient way of declaring constants in a C program. Like the #includes, the #defines are preprocessor control lines. #defines are processed at an early stage of the compilation. They cause the substitution of the named identifier by the associated token string throughout the text of the part of the program which follows the #define. Exception: not if the identifier is embedded inside a comment, a quoted string, or a char constant. Typically, a #define is of the form:

#define IDENTIFIER identifier_will_be_replaced_by_this_stuff

In Program 16 above wherever PI occurs in the program, it is replaced by 3.1415928. (There is a longstanding tradition in C for writing names for MACRO constants in upercases characters, and we will follow this practice.)

Like the #include, #defines are also not terminated by a semicolon; if they were, the semicolon would become part of the replacement string, and this could cause syntactical errors. For example consider the following program:

/*@ Program 17; file name:unit2-progl7.c */
#define PI 3.1415928;  /* Error! */
int main()  
{  /* finds volume of cone, height H, base radius R */
    double H = 1.23, R = 2.34, cone_volume;
    cone_volume = PI * R * R * H / 3;
    return (0);
}

When the replacement for PI is made, the assignment for cone_volume takes the form:

cone_volume = 3.1415928; *2.34 *2.34 *1.23 /3;

which cannot be compiled. For precisely the same reason the assignment operator = cannot occur in a MACRO definition.

A #defined quantity is not a variable and its value cannot be modified by an assignment. Though the MACRO definition for PI has been placed before main() in Program 17, this is not a requirement: it may occur anywhere in the program before it is referenced. One great convenience afforded by MACRO definitions is that if the token string representing the value of a #defined quantity has to be changed, it need be changed only once, in the MACRO itself. When the program is compiled, the changed value will be recorded in every occurrence of the quantity following the #definition. So if you wish to make a computation in which the value of PI must be accurate to 16 places of decimals, just change it where it's #defined.
Introduction to the C Programming Language

ANSI C has another floating point type called long double which has at least as large a number of significant figures, and as large a range of allowable exponents as double. The system file float.h contains constants pertaining to floating point arithmetic. Some of these for gcc on 32 bit Linux are:

- DBL_DIG: decimal digits of precision = 15
- FLT_DIG: decimal digits of precision = 6
- LDBL_DIG: decimal digits of precision = 18
- DBL_MANT_DIG: bits to hold the mantissa = 53
- FLT_MANT_DIG: bits to hold the mantissa = 24
- LDBL_MANT_DIG: bits to hold the mantissa = 64
- DBL_MAX_10_EXP: maximum exponent values = 308
- FLT_MAX_10_EXP: maximum exponent values = 38
- LDBL_MAX_10_EXP: maximum exponent values = 4932
- DBL_MIN_10_EXP: minimum exponent values = -307
- FLT_MIN_10_EXP: minimum exponent values = -37
- LDBL_MIN_10_EXP: minimum exponent values = -4931

You can always print these values for your machine by writing a small C program similar to the one we wrote for integers.

Here is a version of Simpson rule that uses #define to define the integrand.

```c
#include <stdio.h>
#include <math.h>
#define f(x) (1/(1+(x)*(x))) /* You can change this */
/* Note that the previous line does not contain a =
   between f(x) and 1/(1+(x)*(x)) */
int main()
{
  float h = 1 / 4.0;
  printf("The value of the integral is \n%f\n", (h / 3.0) * (f(0) + 4 * (f(h)) + /*Continued*/
(f(3 * h)) + 2 * f(2 * h) + f(4 * h))); /*Continued*/
  return (0);
}
```

ANSI C provides the const declaration for items whose values should not change in a program:

```c
const int dozen = 12;
```

The keyword const lets the programmer specify the type explicitly in contrast to the #define, where the type is deduced from the definition. In addition, ANSI C has a modifier volatile, to explicitly indicate variables whose values may change, and which must be accessed for a possibly changed value whenever they are referenced.
Here are some exercises to give you practice in using variables of type float.

E11) Write a C program to compute the volume of a sphere of radius 10. The volume is given by the formula:

\[ V = \frac{4}{3} \pi R^3 \]

where \( R \) is the radius of the sphere.

In the next section, we will discuss some other data types called enumerated types.

---

2.6 ENUMERATED TYPES

In addition to these four types of program variables, C allows additional user-defined variable types, called `enum` (from enumerated) types. Consider the following declaration:

```c
enum grades
{
  F, D, C_MINUS, C, C_PLUS, B_MINUS, B, B_PLUS,
  A_MINUS, A, A_PLUS
} result;
```

This declaration makes the variable `result` an enumerated type, namely `grades`, which is called the **tag** of the type. The tag is a reference to the enumerated type, and may be used to declare other variables of type `enum grades`:

```c
enum grades final_result;
```

The variables `result` and `final_result` can only be assigned an of the values `F`, `D`, ..., `A_PLUS`, and no others. These eleven enumerators have the consecutive integer values 0 (for `F`) through 10 (for `A_PLUS`); so that if you assign the value `A_PLUS` to `result` and later examine it, it will be found to be 10 (and not the string "A_PLUS"). Though the C compiler stores enumerated values as integer constants, `enum` variables are a distinct type, and they should not be thought of as `ints`. [In ANSI C the enumerators are `int` constants.]

In an `enum` list any enumerator may be specified an integer value different from the constant associated with it by default. The enumerator to its right gets a value 1 greater, and further down the list, each enumerator becomes 1 more than the preceding.

```c
enum flavours
{
  sweet, sour, salty = 6, pungent, hot, bitter
} pickles;
```

In the declaration above `sweet` and `sour` have the values 0 and 1 respectively, while `pungent`, `hot` and `bitter` are 7, 8 and 9 in that order.

---

2.7 THE typedef STATEMENT

The `typedef` statement is used when one wants to refer to a variable type by an alternative name, or alias. This often makes a great convenience for the programmer: suppose you are writing a program to keep track of all the cutlery—spoons, forks, knives and serving lades in a restaurant. Then, the statement:

```c
typedef int cutlery;
```
will enable you to declare:

```c
cutlery spoons, forks, knives, serving_ladles;
```

which may be more meaningful than:

```c
int spoons, forks, knives, serving_ladles;
```

cutlery becomes an alias for int. typedef does no more than rename an existing type.

---

E12) Name the types of C variables you would choose to represent the following quantities—char, int, float, double or enum:

1) The velocity of sound in air, approximately 750 miles/hour.

2) The velocity of light—in vacuum, 2.997925E10 cm/sec.

3) The number of seconds in 400 years.

4) The punctuation symbols of the English alphabet.

5) The months of the year.

6) The population of the city of Delhi, approximately 9 million.

7) The population of planet Earth, approximately 5.4 million.

8) The value of Avogadro’s number, 6.0248E23.

9) The value of Planck’s constant, 6.61E-27.

10) 6.594,126,820,000,000,000,000, which is the approximate value of the mass of the earth, in tons.

11) The colours of the rainbow.

12) The II class train fare between any two cities in India.

13) The seat capacity of a Boeing 747.

14) The vowels of the English alphabet.

15) The days of the week.

16) The savings bank balance of Rs. 12,345.67.

17) The savings bank balance of Rs. 12,345,689.10.

18) The square-root of 6 correct to 6 significant figures.

19) The cube-root of 15 correct to 15 significant figures.

---

So far we have merrily given names to variables in C programs without bothering about their validity. However, there are some rules governing variable names. We will discuss these rules in the next section.

### 2.8 IDENTIFIERS

We have already seen several examples of C identifiers, or names for program variables (more precisely, the name of storage locations): newline, octal_num, apples, volt, etc. Identifiers for variables and constants, as well as for functions, are sequences of characters chosen from the set {A-Z, a-z, 0-9, _}, of which the first character must not be a digit. C is a case sensitive language, so that ALFA and ALFa are different identifiers, as are main() and Main(). The underscore character (_) should not be used as the first character of a variable name because several compiler defined identifiers in the standard C library have the underscore for the beginning character, and inadvertently duplicated names can cause “definition conflicts”. Identifiers may be any reasonable length; generally 8-10 characters should suffice, though certain compilers may allow for very much longer names (of up to 63 characters).
ANSI C: According to ANSI C standards, at least first 31 characters of internal variables are significant. (All the variables that we have seen so far are internal variables only.) In other words, two identifiers must be regarded as different if at least one of the first 31 of their characters are different. We will see what are external variables in the next block. For external variables, at least 6 characters are significant.

C has a list of keywords e.g. int, continue, etc. which cannot be used in any context other than that predefined in the language. (This implies, for example, that you can't have a program variable named int. A list of keywords is appended below; take care that you do not choose identifiers from this list. Your programs will not compile. Indeed one should consistently follow the practice of choosing names for variables which indicate the roles of those variables in the program. For example, the identifier saving_balance in a program that processes savings-bank balances is clearly a better choice for representing the variables than is asdf.

<table>
<thead>
<tr>
<th>C Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>extern</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>for</td>
</tr>
<tr>
<td>goto</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>register</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>signed</td>
</tr>
<tr>
<td>sizeof</td>
</tr>
<tr>
<td>static</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>typedef</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>volatile</td>
</tr>
<tr>
<td>while</td>
</tr>
</tbody>
</table>

E13) The output of the program below on an ASCII machine was the alphabetical character e. What is the ASCII decimal equivalent of d?

```c
/* Program 2.18; File name:unit2-progl8.c */
#include <stdio.h>
int main()
{
    char a, b, c = 'd';
    b = c / 10;
    a = b * b + 1;
    putchar(a);
    return (0);
}
```

E14) State the output of programs 2.19 and 2.20, and verify your results on a computer:

```c
/* Program 2.19; File name:unit2-progl9.c */
#include <stdio.h>
int main()
{
    int alpha = 077, beta = 0xab, gamma = 123, q;
    q = alpha + beta - gamma;
    printf("%d\n", q);
    q = beta / alpha;
    printf("%d\n", q);
    q = beta % gamma;
    printf("%d\n", q);
    q = beta / (alpha + gamma);
    printf("%d\n", q);
    return (0);
}
```
E15) Write C programs to verify whether:

\[36^2 + 37^2 + 38^2 + 39^2 + 40^2 = 41^2 + 42^2 + 43^2 + 44^2\]
\[23^3 + 24^3 + 25^3 = 204^2\]
\[5^8 + 12^8 + 13^8 = 59^4 + 120^4 + 179^4\]

E16) The Fibonacci numbers \(F_1, F_2, F_3, \ldots, F_n\) are defined by the relations:

\[F_1 = 1\]
\[F_2 = 1\]
\[F_n = F_{n-1} + F_{n-2}, n > 2\]

\(F_3\) and successive numbers of the series are obtained by adding the preceding two numbers. For large \(n\) the ratio of two consecutive Fibonacci numbers is approximately 0.618033989. Given that \(F(100)\) is

354, 224, 848, 179, 261, 915, 075(21 digits)

Write a C program to find approximations to \(F(99)\) and \(F(101)\).

E17) The Lucas numbers are defined by the same recurrence relation as the Fibonacci's, where \(L_1 = 1\) but \(L_2 = 3\). Write a C program to print the first 10 Lucas numbers.

E18) The first large number to be factorised by a mechanical machine was one of 19 digits:

1, 537, 228, 672, 093, 361, 419.

The two prime factors found (which were known beforehand) were 529, 510, 939 and 2, 903, 110, 321. Writing about that event later in Scripta Mathematica 1 (1933) [quoted by Malcolm E Lines in Think of a Number, Pub. Adam Hilger, 1990] Professor D. N. Lehmer (who with his son had invented the machine) remarked,
"It would have surprised you to see the excitement in the group of professors and their wives, as they gathered around a table in the laboratory to discuss, over coffee, the mysterious and uncanny powers of this curious machine. It was agreed that it would be unsportsmanlike to use it on small numbers such as could be handled by factor tables and the like, but to reserve it for numbers which lurk as it were, in other galaxies than ours, outside the range of ordinary telescopes."

Write a C program to determine, to the best extent you can, whether the two factors found are indeed correct.

We close the Unit here. In the next section, we will summarise the Unit.

2.9 SUMMARY

The fundamental data types of C are declared through the seven keywords: int, long, short, unsigned, char, float and double. (The keywords signed and long double are ANSI C extensions.)

1. Integers: ranges are in general machine dependent, but in any implementation a short will be at most as long as an int, and an int will be no longer than long. Unsigned integers have zero or positive values only.

2. Characters: char variables are used to represent byte-sized integers and textual symbols.

3. Floating point numbers: may be single or double precision numbers with a decimal point. Unless specified to the contrary (in ANSI C), single precision floats are automatically converted to double in a computation. double variables allow a larger number of significant figures, and a larger range of exponents than floats. getchar() gets a keystroke from the keyboard; putchar() deposits its character argument on the monitor.

4. printf() uses the following format conversion characters to print variables:
   d   decimal integers
   u   unsigned integers
   o   octal integers
   l   long
   x   hex integers, lowercase
   X   hex integers, uppercase
   f   floating point numbers
   e   floating point numbers in exponential format, lowercase e
   E   floating point numbers in exponential format, uppercase E
   g   floating point numbers in the shorter of f or e format
   G   floating point numbers in the shorter of f or E format e single characters

The modifier # with x or X causes a leading 0x or 0X to appear in the output of hexadecimal values; scanf() uses %If to read doubles.

2.10 Solutions/Answers

E1) Here x/y gives the quotient of x when divided by y. When x = 60, y = 15, x/y is 4; when x = 70, y = 15 x/y is 4; when x = 75, y = 15, x/y is 5 etc.

E2) Here x % y gives remainder when x is divided by y if y < y. It gives y if x < y and 0 if x = y.
Introduction to the C Programming Language

E3) Here is the program:

```c
#include<stdio.h>

int main()
{
    unsigned cheque = 54321;
    int seconds, minutes, hours, days;
    long time = 1234567890L;  /*seconds */
    printf("I've waited a long time (%ld seconds)\n", time);
    printf("for my cheque (for Rs.%u/-), and now\n", cheque);
    printf("I find it\'s unsigned!\n");
    printf("That's a wait of: \n");
    seconds = time % 60;
    time = time / 60;
    minutes = time % 60;
    time = time / 60;
    hours = time % 24;
    days = time / 24;
    printf("%d days\n", days);
    printf("%d hours\n", hours);
    printf("%d minutes\n", minutes);
    printf("%d seconds\n", seconds);
    return (0);
}
```

E4) Here is the example program:

```c
#include<stdio.h>

int main()
{
    /* p is in octal notation, q is in hex. */
    int p = 012345, q = 0x1234;
    long octal_num = 012345670L, hex_num = 0x7BCDEF89L;
    printf("Value of p is %#o\n", p);
    printf("Value of q is %#lx\n", q);
    printf("Value of octal num is %#lo\n", octal_num);
    printf("Value of hex num is %#lx\n", hex_num);
    return (0);
}
```

E5) The following is the output on a 32-bit Linux machine when the program is compiled with gcc:

```
x = 5
x = 25
x = 625
x = 390625
x = -2030932031
```

E6) /*Answer to exercise 6; Filename: unit2-printlimitsfull.c*/

```c
#include<stdio.h>
#include<limits.h>

int main()
{
    printf("Number of bits in a char is: %d\n", CHAR_BIT);
    printf("Maximum value of char is %d\n", CHAR_MAX);
    printf("Minimum value of char is %d\n", CHAR_MIN);
    printf("Maximum value of int is %d\n", INT_MAX);
    printf("Minimum value of int is %d\n", INT_MIN);
    printf("Maximum value of long is %ld\n", LONG_MAX);
    printf("Minimum value of long is %ld\n", LONG_MIN);
    printf("Maximum value of signed 
```
char is %d: \n", CHAR_MAX); /*Continued*/
printf("Minimum value of signed is %d: \n", CHAR_MIN);
printf("Maximum value of unsigned \n"
char is %d: \n", UCHAR_MAX); /*Continued*/
printf("Maximum value of unsigned \n"
int is %u: \n", UINT_MAX); /*Continued*/
printf("Maximum value of unsigned \n"
long is %lu: \n", ULONG_MAX); /*Continued*/
return (0);
}
E7) /*Program that checks for overflow.
File: unit2-ex7ans.c*/
#include <stdio.h>
#include <limits.h>
int main() {
  unsigned long x, y;
  printf("Enter the value of x...
");
  scanf("%lu", &x);
  printf("Enter the value of y...
");
  scanf("%lu", &y);
  if (x > ULONG_MAX - y)
    printf("The sum of x and y exceeds
the limit of ULONG_MAX\n"); /*Continued*/
  else
    printf("\nThe sum of x and y is %lu\n", x + y);
  return 0;
}
E8) /* Program ans-ex7; file name: prog13-ex7.c*/
#include <stdio.h>
int main() {
  int alpha = 'A', beta;
  char gamma = 122;
  unsigned char delta = alpha - gamma;
  beta = gamma - alpha;
  printf("beta seen as an int is: %d\n", beta);
  printf("beta seen as a char is: %c\n", beta);
  printf("delta as an int %d\n", delta);
  printf("delta as an unsigned int %u\n", delta);
  printf("delta as a char is %c\n", delta);
  return (0);
}
The output is
beta seen as an int is: 57
beta seen as a char is: 9
delta as an int 199
delta as an unsigned int 199
delta as a char is
Introduction to the C Programming Language

```c
i = getchar();
printf("The decimal equivalent is %d. ", i);
return 0;
}
```

E10) /* Program to calculate simple interest. 
File name: unit2-ansex10.c */
#include <stdio.h>
int main()
{
    double principal, rate_of_interest, period, interest;
    printf("This program calculates simple interest\n");
    printf("Enter the principal\n");
    scanf("%lf", &principal);
    printf("Enter the rate of interest: \n");
    scanf("%lf", &rate_of_interest);
    printf("Enter the period in months: \n");
    scanf("%lf", &period);
    interest=(principal * rate_of_interest * period) / 100;
    printf("The simple interest is \n");
    printf("%lf\n", interest);
    return (0);
}
```

E11) /* Program to calculate the volume of a sphere 
File name: unit2-ansex11.c */
#include <stdio.h>
#define PI 3.1415928
int main()
{
    double radius;
    printf("Enter the radius of the sphere: \n");
    scanf("%lf", &radius);
    printf("The volume of the sphere is: \n");
    printf("%lf", 4*PI*radius*radius*radius/3);
    return (0);
}
```

E12)
1) float
2) float
3) It is an integer type, but none of the integer types is big enough. So, we have to use float.
4) char
5) enum
6) int
7) float
8) float
9) float
10) double

E13) 100

E14) The output from program 19 is
```
2688
43
42
14
```
Hello, world!

E15) /*Answer to exercise 14; File name unit2-ans-ex14.c*/
   #include <stdio.h>
   long sq(int x);
   long cu(int x);
   main()
   {
      long ans, ans1;
      ans = sq(36) + sq(37) + sq(38) + sq(39) + sq(40);
      ans1 = sq(41) + sq(42) + sq(43) + sq(44);
      printf("%ld", ans - ans1);
      ans = cu(23) + cu(24) + cu(25);
      ans1 = sq(204);
      printf("\n%ld", ans - ans1);
      ans = cu(sq(5)) * sq(5) + cu(sq(12)) * sq(12) + cu(sq(13)) * sq(13); /*Continued*/
      ans1 = sq(sq(59)) + sq(sq(120)) + sq(sq(179));
      printf("\n%ld", ans - ans1);
      return 0;
   }
   long sq(int x)
   {
      return x * x;
   }
   long cu(int x)
   {
      return x * x * x;
   }

E16) /*Answer to exercise 15.*/
   /*File name: unit2-ans-ex15.c*/
   #include <stdio.h>
   int main()
   {
      long double f99, f100 = 354224848179261915075.0L, f101;
      long double ratio = 0.618033989L;
      printf("\n%f100=%Lf\n", f100);
      f99 = f100 * ratio;
      f101 = f100 / ratio;
      printf("f99 = %Lf\n f100 = %Lf\n f101 = %Lf\n", f99, f100, f101);/*Continued*/
      printf("f101-f100-f99=%Lf", f101 - f99 - f100);
      return 0;
   }

   Note the %Lf format modifier for long double. %Le is also allowed for printing
   long double in exponential notation.

E17) /*Answer to exercise 18; File name: unit2-ansex18.c*/
   #include <stdio.h>
   int main()
   {
      long double prime_1 = 529510939.0L;
      long double prime_2 = 2903110321.0L, product;
      product = prime_1 * prime_2;
      printf("The product of the primes is %Lf", product);
      return 0;
   }

   Does the program give the correct answer?
### ASCII Table

<table>
<thead>
<tr>
<th>Char</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Oct</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(nul)</td>
<td>0</td>
<td>0000</td>
<td>0x00</td>
<td>(sp)</td>
<td>32</td>
<td>0040</td>
<td>0x20</td>
<td>@</td>
<td>64</td>
<td>0100</td>
<td>0x40</td>
</tr>
<tr>
<td>(soh)</td>
<td>1</td>
<td>0001</td>
<td>0x01</td>
<td>&quot;</td>
<td>33</td>
<td>0041</td>
<td>0x21</td>
<td>A</td>
<td>65</td>
<td>0101</td>
<td>0x41</td>
</tr>
<tr>
<td>(stx)</td>
<td>2</td>
<td>0002</td>
<td>0x02</td>
<td>&quot;n</td>
<td>34</td>
<td>0042</td>
<td>0x22</td>
<td>B</td>
<td>66</td>
<td>0102</td>
<td>0x42</td>
</tr>
<tr>
<td>(etx)</td>
<td>3</td>
<td>0003</td>
<td>0x03</td>
<td>#</td>
<td>35</td>
<td>0043</td>
<td>0x23</td>
<td>C</td>
<td>67</td>
<td>0103</td>
<td>0x43</td>
</tr>
<tr>
<td>(eot)</td>
<td>4</td>
<td>0004</td>
<td>0x04</td>
<td>$</td>
<td>36</td>
<td>0044</td>
<td>0x24</td>
<td>D</td>
<td>68</td>
<td>0104</td>
<td>0x44</td>
</tr>
<tr>
<td>(enq)</td>
<td>5</td>
<td>0005</td>
<td>0x05</td>
<td>%</td>
<td>37</td>
<td>0045</td>
<td>0x25</td>
<td>E</td>
<td>69</td>
<td>0105</td>
<td>0x45</td>
</tr>
<tr>
<td>(ack)</td>
<td>6</td>
<td>0006</td>
<td>0x06</td>
<td>&amp;</td>
<td>38</td>
<td>0046</td>
<td>0x26</td>
<td>F</td>
<td>70</td>
<td>0106</td>
<td>0x46</td>
</tr>
<tr>
<td>(bel)</td>
<td>7</td>
<td>0007</td>
<td>0x07</td>
<td>‘</td>
<td>39</td>
<td>0047</td>
<td>0x27</td>
<td>G</td>
<td>71</td>
<td>0107</td>
<td>0x47</td>
</tr>
<tr>
<td>(bs)</td>
<td>8</td>
<td>0010</td>
<td>0x08</td>
<td>(</td>
<td>40</td>
<td>0050</td>
<td>0x28</td>
<td>H</td>
<td>72</td>
<td>0110</td>
<td>0x48</td>
</tr>
<tr>
<td>(ht)</td>
<td>9</td>
<td>0011</td>
<td>0x09</td>
<td>)</td>
<td>41</td>
<td>0051</td>
<td>0x29</td>
<td>I</td>
<td>73</td>
<td>0111</td>
<td>0x49</td>
</tr>
<tr>
<td>(nl)</td>
<td>10</td>
<td>0012</td>
<td>0x0a</td>
<td>*</td>
<td>42</td>
<td>0052</td>
<td>0x2a</td>
<td>J</td>
<td>74</td>
<td>0112</td>
<td>0x4a</td>
</tr>
<tr>
<td>(vt)</td>
<td>11</td>
<td>0013</td>
<td>0x0b</td>
<td>+</td>
<td>43</td>
<td>0053</td>
<td>0x2b</td>
<td>K</td>
<td>75</td>
<td>0113</td>
<td>0x4b</td>
</tr>
<tr>
<td>(np)</td>
<td>12</td>
<td>0014</td>
<td>0x0c</td>
<td>.</td>
<td>44</td>
<td>0054</td>
<td>0x2c</td>
<td>L</td>
<td>76</td>
<td>0114</td>
<td>0x4c</td>
</tr>
<tr>
<td>(cr)</td>
<td>13</td>
<td>0015</td>
<td>0x0d</td>
<td>-</td>
<td>45</td>
<td>0055</td>
<td>0x2d</td>
<td>M</td>
<td>77</td>
<td>0115</td>
<td>0x4d</td>
</tr>
<tr>
<td>(so)</td>
<td>14</td>
<td>0016</td>
<td>0x0e</td>
<td>.</td>
<td>46</td>
<td>0056</td>
<td>0x2e</td>
<td>N</td>
<td>78</td>
<td>0116</td>
<td>0x4e</td>
</tr>
<tr>
<td>(si)</td>
<td>15</td>
<td>0017</td>
<td>0x0f</td>
<td>/</td>
<td>47</td>
<td>0057</td>
<td>0x2f</td>
<td>O</td>
<td>79</td>
<td>0117</td>
<td>0x4f</td>
</tr>
<tr>
<td>(dle)</td>
<td>16</td>
<td>0020</td>
<td>0x10</td>
<td>0</td>
<td>48</td>
<td>0060</td>
<td>0x30</td>
<td>P</td>
<td>80</td>
<td>0120</td>
<td>0x50</td>
</tr>
<tr>
<td>(dc1)</td>
<td>17</td>
<td>0021</td>
<td>0x11</td>
<td>1</td>
<td>49</td>
<td>0061</td>
<td>0x31</td>
<td>Q</td>
<td>81</td>
<td>0121</td>
<td>0x51</td>
</tr>
<tr>
<td>(dc2)</td>
<td>18</td>
<td>0022</td>
<td>0x12</td>
<td>2</td>
<td>50</td>
<td>0062</td>
<td>0x32</td>
<td>R</td>
<td>82</td>
<td>0122</td>
<td>0x52</td>
</tr>
<tr>
<td>(dc3)</td>
<td>19</td>
<td>0023</td>
<td>0x13</td>
<td>3</td>
<td>51</td>
<td>0063</td>
<td>0x33</td>
<td>S</td>
<td>83</td>
<td>0123</td>
<td>0x53</td>
</tr>
<tr>
<td>(dc4)</td>
<td>20</td>
<td>0024</td>
<td>0x14</td>
<td>4</td>
<td>52</td>
<td>0064</td>
<td>0x34</td>
<td>T</td>
<td>84</td>
<td>0124</td>
<td>0x54</td>
</tr>
<tr>
<td>(nak)</td>
<td>21</td>
<td>0025</td>
<td>0x15</td>
<td>5</td>
<td>53</td>
<td>0065</td>
<td>0x35</td>
<td>U</td>
<td>85</td>
<td>0125</td>
<td>0x55</td>
</tr>
<tr>
<td>(syn)</td>
<td>22</td>
<td>0026</td>
<td>0x16</td>
<td>6</td>
<td>54</td>
<td>0066</td>
<td>0x36</td>
<td>V</td>
<td>86</td>
<td>0126</td>
<td>0x56</td>
</tr>
<tr>
<td>(etb)</td>
<td>23</td>
<td>0027</td>
<td>0x17</td>
<td>7</td>
<td>55</td>
<td>0067</td>
<td>0x37</td>
<td>W</td>
<td>87</td>
<td>0127</td>
<td>0x57</td>
</tr>
<tr>
<td>(can)</td>
<td>24</td>
<td>0030</td>
<td>0x18</td>
<td>8</td>
<td>56</td>
<td>0070</td>
<td>0x38</td>
<td>X</td>
<td>88</td>
<td>0130</td>
<td>0x58</td>
</tr>
<tr>
<td>(em)</td>
<td>25</td>
<td>0031</td>
<td>0x19</td>
<td>9</td>
<td>57</td>
<td>0071</td>
<td>0x39</td>
<td>Y</td>
<td>89</td>
<td>0131</td>
<td>0x59</td>
</tr>
<tr>
<td>(sub)</td>
<td>26</td>
<td>0032</td>
<td>0x1a</td>
<td>:</td>
<td>58</td>
<td>0072</td>
<td>0x3a</td>
<td>Z</td>
<td>90</td>
<td>0132</td>
<td>0x5a</td>
</tr>
<tr>
<td>(asc)</td>
<td>27</td>
<td>0033</td>
<td>0x1b</td>
<td>;</td>
<td>59</td>
<td>0073</td>
<td>0x3b</td>
<td>[</td>
<td>91</td>
<td>0133</td>
<td>0x5b</td>
</tr>
<tr>
<td>(fs)</td>
<td>28</td>
<td>0034</td>
<td>0x1c</td>
<td>&lt;</td>
<td>60</td>
<td>0074</td>
<td>0x3c</td>
<td>\</td>
<td>92</td>
<td>0134</td>
<td>0x5c</td>
</tr>
<tr>
<td>(gs)</td>
<td>29</td>
<td>0035</td>
<td>0x1d</td>
<td>=</td>
<td>61</td>
<td>0075</td>
<td>0x3d</td>
<td>]</td>
<td>93</td>
<td>0135</td>
<td>0x5d</td>
</tr>
<tr>
<td>(rs)</td>
<td>30</td>
<td>0036</td>
<td>0x1e</td>
<td>&gt;</td>
<td>62</td>
<td>0076</td>
<td>0x3e</td>
<td>^</td>
<td>94</td>
<td>0136</td>
<td>0x5e</td>
</tr>
<tr>
<td>(us)</td>
<td>31</td>
<td>0037</td>
<td>0x1f</td>
<td>?</td>
<td>63</td>
<td>0077</td>
<td>0x3f</td>
<td>_</td>
<td>95</td>
<td>0137</td>
<td>0x5f</td>
</tr>
</tbody>
</table>

Note that the characters from 0–31 and character 127 are non-printing characters and character 32 is the space.